PATENT

CERTIFICATE OF MAILING VIA EXPRESS MAIL
37 C.F.R. 1.10

I HEREBY CERTIFY THAT I HAVE INFORMATION AND A REASONABLE BASIS FOR BELIEF THAT THIS CORRESPONDENCE IS BEING DEPOSITED WITH THE UNITED STATES POSTAL SERVICE AS EXPRESS MAIL POST OFFICE TO ADDRESSEE ON THE DATE INDICATED BELOW AND IS ADDRESSED TO:

MAIL STOP PROVISIONAL PATENT APPLICATION
HONORABLE COMMISSIONER FOR PATENTS
P. O. BOX 1450
ALEXANDRIA, VA 22313.1450

Ronald L. Chichester

RONALD L. CHICHESTER REGISTRATION NO. 36,765

DATE OF MAILING: MARCH 9, 2004
EXPRESS MAIL LABEL: EV339226309US

APPLICATION FOR LETTERS PATENT

FOR

MICROCONTROLLER INSTRUCTION SET

Inventors: Edward Brian Boles, Rodney Jay Drake, Darrel Ray Johansen, Sumit K. Mitra, Randy Yach, James Grosbach, Joshua M. Conner and Joseph W. Triece

ASSIGNEE:	**Microchip Technology Incorporated**
ATTORNEY:	**Ronald L. Chichester of Baker Botts L.L.P.**
ATTORNEY DOCKET NO.:	**068354.1410**
CLIENT REFERENCE NO.:	**MTI-2094.US.0**

Microcontroller Instruction Set

SPECIFICATION

CROSS REFERENCE TO RELATED APPLICATION

[0001] This application is a continuation-in-part of U.S. Serial Number 09/280,112 that was filed on March 26, 1999 by the same title and to the same inventors as the present application. This application is related to the following applications: U.S. Patent Nos. 6,055,211 for "FORCE PAGE ZERO PAGING SCHEME FOR MICROCONTROLLERS USING DATA ACCESS MEMORY" by Randy L. Yach, et al.; 5,905,880 for "ROBUST MULTIPLE WORK INSTRUCTION AND METHOD THEREFOR" by Rodney J. Drake, et al.; 6,192,463 for "PROCESSOR ARCHITECTURE SCHEME WHICH USES VIRTUAL ADDRESS REGISTERS TO IMPLEMENT DIFFERENT ADDRESSING MODES AND METHOD THEREFOR" by Sumit Mitra, et al. ; 6,243,798 for "COMPUTER SYSTEM FOR ALLOWING A TWO WORD INSTRUCTION TO BE EXECUTED IN THE SAME NUMBER OF CYCLES AS A SINGLE WORD JUMP INSTRUCTION" by Rodney J. Drake, et al. ; 6,029,241 entitled "PROCESSOR ARCHITECTURE SCHEME HAVING MULTIPLE BANK ADDRESS OVERRIDE SOURCES FOR SUPPLYING ADDRESS VALUES AND METHOD THEREFORE" by Igor Wojewoda, Sumit Mitra, and Rodney J. Drake; 6,098,160 for "DATA POINTER FOR OUTPUTTING INDIRECT ADDRESSING MODE ADDRESSES WITHIN A SINGLE CYCLE AND METHOD THEREFOR" by Rodney J. Drake, et al.; 5,958,039 for "MASTER-SLAVE LATCHES AND POST INCREMENT/DECREMENT OPERATION" by Allen, et al.; and 5,987,583 for "PROCESSOR ARCHITECTURE SCHEME AND INSTRUCTION SET FOR MAXIMIZING AVAILABLE OPCODES AND ADDRESSING SELECTION MODES" by Triece, et al. which are hereby incorporated herein by reference for all purposes.

BACKGROUND OF THE INVENTION

FIELD OF THE INVENTION

[0002] The present invention relates to microcontrollers and, more specifically, the present invention relates to opcode instructions that are gathered into an instruction set which are used to manipulate the behavior of the microcontroller.

DESCRIPTION OF THE RELATED TECHNOLOGY

[0003] Microcontroller units (MCU) have been used in the manufacturing and electrical industries for many years. Figure 1 shows a typical core memory bus arrangement for mid-range MCU devices. In many cases, microcontrollers utilize reduced instruction set computing (RISC) microprocessors. The high performance of some of these devices can be attributed to a number of architectural features commonly found in RISC microprocessors. These features include:

Harvard architecture
Long Word Instructions
Single Word Instructions
Single Cycle Instructions
Instruction Pipelining
Reduced Instruction Set
Register File Architecture
Orthogonal (Symmetric) Instructions

Harvard Architecture:

[0004] As shown in Figure 2, the Harvard architecture has the program memory 26 and data memory 22 as separate memories and are accessed by the CPU 24 from separate buses. This improves bandwidth over traditional von Neumann architecture (shown in Figure 3) in which program and data are fetched by the CPU 34 from the same memory 36 using the same bus. To execute an instruction, a von Neumann machine must make one or more (generally more) accesses across the 8-bit bus to fetch the instruction. Then data may need to be fetched,

operated on, and possibly written. As can be seen from this description, that bus can become extremely congested.

[0005] In contrast to the von Neumann machine, under the Harvard architecture, all 14 bits of the instruction are fetched in a single instruction cycle. Thus, under the Harvard architecture, while the program memory is being accessed, the data memory is on an independent bus and can be read and written. These separated buses allow one instruction to execute while the next instruction is being fetched.

Long Word Instructions:

[0006] Long word instructions have a wider (more bits) instruction bus than the 8-bit Data Memory Bus. This is possible because the two buses are separate. This further allows instructions to be sized differently than the 8-bit wide data word which allows a more efficient use of the program memory, since the program memory width is optimized to the architectural requirements.

Single Word Instructions:

[0007] Single Word instruction opcodes are 14-bits wide making it possible to have all single word instructions. A 14-bit wide program memory access bus fetches a 14-bit instruction in a single cycle. With single word instructions, the number of words of program memory locations equals the number of instructions for the device. This means that all locations are valid instructions. Typically in the von Neumann architecture (shown in Figure 3), most instructions are multi-byte. In general however, a device with 4-KBytes of program memory would allow approximately 2K of instructions. This 2:1 ratio is generalized and dependent on the application code. Since each instruction may take multiple bytes, there is no assurance that each location is a valid instruction.

Instruction Pipeline:

[0008] The instruction pipeline is a two-stage pipeline which overlaps the fetch and execution of instructions. The fetch of the instruction takes one machine cycle ("TCY"), while the execution takes another TCY. However, due to the overlap of the fetch of current instruction and execution of previous instruction, an instruction is fetched and another instruction is executed every single TCY.

Single Cycle Instructions:

[0009] With the Program Memory bus being 14-bits wide, the entire instruction is fetched in a single TCY. The instruction contains all the information required and is executed in a single cycle. There may be a one-cycle delay in execution if the result of the instruction modified the contents of the Program Counter. This requires that the pipeline be flushed and a new instruction fetched.

Reduced Instruction Set:

[0010] When an instruction set is well designed and highly orthogonal (symmetric), fewer instructions are required to perform all needed tasks. With fewer instructions, the whole set can be more rapidly learned.

Register File Architecture:

[0011] The register files/data memory can be directly or indirectly addressed. All special function registers, including the program counter, are mapped in the data memory.

Orthogonal (Symmetric) Instructions:

[0012] Orthogonal instructions make it possible to carry out any operation on any register using any addressing mode. This symmetrical nature and lack of "special instructions" make programming simple yet efficient. In addition, the learning curve is reduced significantly. The mid-range instruction set uses only two non-register oriented instructions, which are used for two

of the cores features. One is the SLEEP instruction that places the device into the lowest power use mode. The other is the CLRWDT instruction which verifies the chip is operating properly by preventing the on-chip Watchdog Timer (WDT) from overflowing and resetting the device.

Clocking Scheme/Instruction Cycle:

[0013] The clock input (from OSC1) is internally divided by four to generate four non-overlapping quadrature clocks, namely Q1, Q2, Q3, and Q4. Internally, the program counter (PC) is incremented every Q1, and the instruction is fetched from the program memory and latched into the instruction register in Q4. The instruction is decoded and executed during the following Q1 through Q4. The clocks and instruction execution flow are illustrated in Figures 4 and 5.

Instruction Flow/Pipelining:

[0014] An "Instruction Cycle" consists of four Q cycles (Q1, Q2, Q3, and Q4) as shown in Figures 4 that comprise the TCY as shown in Figures 4 and 5. Note that in Figure 5, all instructions are performed in a single cycle, except for any program branches. Program branches take two cycles because the fetch instruction is "flushed" from the pipeline while the new instruction is being fetched and then executed.

[0015] Fetch takes one instruction cycle while decode and execute takes another instruction cycle. However, due to Pipelining, each instruction effectively executes in one cycle. If an instruction causes the program counter to change (e.g. GOTO) then an extra cycle is required to complete the instruction (Figure 5). The instruction fetch begins with the program counter incrementing in Q1. In the execution cycle, the fetched instruction is latched into the "Instruction Register (IR)" in cycle Q1. This instruction is then decoded and executed during the Q2, Q3, and Q4 cycles. Data memory is read during Q2 (operand read) and written during Q4 (destination write). Figure 5 shows the operation of the two-stage pipeline for the instruction

sequence shown. At time TCY0, the first instruction is fetched from program memory. During TCY1, the first instruction executes while the second instruction is fetched. During TCY2, the second instruction executes while the third instruction is fetched. During TCY3, the fourth instruction is fetched while the third instruction (CALL SUB_1) is executed. When the third instruction completes execution, the CPU forces the address of instruction four onto the Stack and then changes the Program Counter (PC) to the address of SUB_1. This means that the instruction that was fetched during TCY3 needs to be "flushed" from the pipeline. During TCY4, instruction four is flushed (executed as a NOP) and the instruction at address SUB_1 is fetched. Finally during TCY5, instruction five is executed and the instruction at address SUB_1 + 1 is fetched.

[0016] While the prior art microcontrollers were useful, the various modules could not be emulated. Moreover, the type of microcontroller as described in Figure 1 could not linearize the address space. Finally, the prior art microcontrollers are susceptible to compiler-error problems. What is needed is an apparatus, method, and system for a microcontroller that is capable of linearizing the address space in order to enable modular emulation. There is also a need in the art for reducing compiler errors.

SUMMARY OF THE INVENTION

[0017] The present invention overcomes the above-identified problems as well as other shortcomings and deficiencies of existing technologies by providing a microcontroller instruction set that eliminates many of the compiler errors experienced in the prior art. Moreover, an apparatus and system is provided that enables a linearized address space that makes modular emulation possible.

[0018] The present invention can directly or indirectly address its register files or data memory. All special function registers, including the Program Counter (PC) and Working Register (W), are mapped in the data memory. The present invention has an orthogonal (symmetrical) instruction set that makes it possible to carry out any operation on any register using any addressing mode. This symmetrical nature and lack of 'special optimal situations' make programming with the present invention simple yet efficient. In addition, the learning curve for writing software applications is reduced significantly. One of the present invention's enhancements over the prior art allows two file registers to be used in some two operand instructions. This allows data to be moved directly between two registers without going through the W register; and thus increasing performance and decreasing program memory usage.

[0019] The preferred embodiment of the present invention includes an ALU/W register, a PLA, an 8-bit multiplier, a program counter (PC) with stack, a table latch/table pointer, a ROM latch/IR latch, FSRs, interrupt vectoring circuitry, and most common status registers. Unlike the prior art, the design of the present invention obviates the need for a timer in a separate module, all reset generation circuitry (WDT, POR, BOR, etc.), interrupt flags, enable flags, INTCON registers, RCON registers, configuration bits, device ID word, ID locations, and clock drivers.

[0020] Additional embodiments will be clear to those skilled in the art upon reference to the detailed description and accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

[0021] Figure 1 is a schematic block diagram of a prior art, mid-range microcontroller unit;

[0022] Figure 2 is a schematic block diagram of the prior art Harvard architecture;

[0023] Figure 3 is a schematic block diagram of the prior art von Neumann architecture;

[0024] Figure 4 is a timing diagram of a prior art clock/instruction cycle;

[0025] Figure 5 is a schematic illustration of the execution of multiple instructions;

[0026] Figure 6 is a schematic block diagram of the microcontroller core of the present invention;

[0027] Figure 7 is a timing diagram of the Q cycle activity of the present invention;

[0028] Figure 8 is a timing diagram of the clock/instruction cycle of the present invention;

[0029] Figure 9 is an instruction pipeline flow diagram of the present invention;

[0030] Figure 10 is an instruction pipeline flow diagram of the present invention;

[0031] Figure 11 is an instruction pipeline flow diagram of the present invention;

[0032] Figure 12 is an instruction pipeline flow diagram of the present invention;

[0033] Figure 13 is an instruction pipeline flow diagram of the present invention;

[0034] Figure 14 is an instruction pipeline flow diagram of the present invention;

[0035] Figure 15 is a block diagram of the status register of the present invention;

[0036] Figure 16 is a block diagram of the program counter of the present invention;

[0037] Figure 17 is a block diagram of the program counter of the present invention using the CALL and GOTO instructions;

[0038] Figure 18 is a block diagram of the stack pointer register of the present invention;

[0039] Figure 19 is a block diagram of the top of stack upper register of the present invention;

[0040] Figure 20 is a block diagram of the top of stack high register of the present invention;

[0041] Figure 21 is a block diagram of the top of stack low register of the present invention;

[0042] Figure 22 illustrates the stack reset operation of the present invention;

[0043] Figure 23 illustrates the first CALL on an initialized stack of the present invention;

[0044] Figure 24 illustrates the second consecutive CALL on a stack of the present invention;

[0045] Figure 25 illustrates a 31st and 32nd consecutive CALL on a stack of the present invention;

[0046] Figure 26 illustrates a return POP operation on a stack of the present invention;

[0047] Figure 27 illustrates a stack return pop causing a stack underflow condition within the present invention;

[0048] Figure 28 illustrates a PUSH instruction on a stack of the present invention;

[0049] Figure 29 illustrates a POP instruction on a stack of the present invention;

[0050] Figure 30 is a block diagram of a program memory map and stack of the present invention;

[0051] Figure 31 is a block diagram of the memory map of the present invention;

[0052] Figure 32 is a block diagram of instructions in the memory of the present invention;

[0053] Figure 33 is a block diagram that illustrates the device memory map of the present invention in different program modes;

[0054] Figure 34 is a block diagram describing the MEMCON register of the present invention;

[0055] Figure 35 is a block diagram describing the CONFIG7 configuration byte of the present invention;

[0056] Figure 36 is a schematic block diagram of the 16-bit external memory connection configuration of the present invention;

[0057] Figure 37 is a block diagram of the 8-bit external memory connection configuration of the present invention;

[0058] Figure 38 is a listing of the typical port functions of the present invention;

[0059] Figure 39 is a timing diagram of the external program memory bus in 16-bit mode of the present invention;

[0060] Figure 40 is a timing diagram of the external program memory bus in 8-bit mode of the present invention;

[0061] Figure 41 is a listing of the external bus cycle types of the present invention;

[0062] Figure 42 is a schematic block diagram of the data memory map and the instruction "a" bit of the present invention;

[0063] Figure 43 is a map of the special function register of the present invention;

[0064] Figure 44 is a schematic of the core special function register of the present invention;

[0065] Figure 45 is a continuation of the schematic of the core special function register of Figure 44;

[0066] Figure 46 is a schematic block diagram of the direct short addressing mode of the present invention;

[0067] Figure 47 is a schematic block diagram of the BSR operation of the present invention;

[0068] Figure 48 is a schematic block diagram of the BSR operation of the present invention during emulation/test modes;

[0069] Figure 49 is a schematic block diagram of the direct forced addressing mode of the present invention;

[0070] Figure 50 is a schematic block diagram of the direct forced addressing mode of the present invention;

[0071] Figure 51 is a schematic block diagram of the direct long addressing mode of the present invention;

[0072] Figure 52 is a schematic block diagram of the indirect addressing mode of the present invention;

[0073] Figure 53 is a schematic block diagram of the indirect addressing mode of the present invention;

[0074] Figure 54 is a descriptive listing opcode fields of the present invention;

[0075] Figure 55 is a listing of indirect addressing symbols of the present invention;

[0076] Figure 56 illustrates the general format for the instructions of the present invention;

[0077] Figure 57 is a partial listing of the instruction set of the present invention;

[0078] Figure 58 is a partial listing of the instruction set of the present invention;

[0079] Figure 59 is a partial listing of the instruction set of the present invention;

[0080] Figure 60 is a flowchart for the byte oriented file register operations of the present invention;

[0081] Figure 61 is a flowchart for the byte oriented file register operations (execute) of the present invention;

[0082] Figure 62 is a flowchart for the CLRF, NEGF, SETF (Fetch) instructions of the present invention;

[0083] Figure 63 is a flowchart for the CLRF, NEGF, SETF (Execute) instructions of the present invention;

[0084] Figure 64 is a flowchart for the DECFSZ, DCFSNZ, INCFSZ, ICFSNZ (Fetch) instructions of the present invention;

[0085] Figure 65 is a flowchart for the DECFSZ, DCFSNZ, INCFSZ, ICFSNZ (Fetch) instructions of the present invention;

[0086] Figure 66 is a flowchart for the CPFSEQ, CPFSQT, CPFSLT, and TSTFSZ (Fetch) instructions of the present invention;

[0087] Figure 67 is a flowchart for the CPFSEQ, CPFSQT, CPFSLT, and TSTFSZ (Execute) instructions of the present invention;

[0088] Figure 68 is a flowchart for the MULWF (Fetch) instruction of the present invention;

[0089] Figure 69 is a flowchart for the MULWF (Execute) instruction of the present invention;

[0090] Figure 70 is a flowchart for the MULFF (Fetch) instruction of the present invention;

[0091] Figure 71 is a flowchart for the MULFF (Execute1) instruction of the present invention;

[0092] Figure 72 is a flowchart for the MULFF (Execute2) instruction of the present invention;

[0093] Fgure 73 is a flowchart for the BCF, BSF, BTG (Fetch) instructions of the present invention;

[0094] Figure 74 is a flowchart for the BCF, BSF, BTG (Fetch) instructions of the present invention;

[0095] Figure 75 is a flowchart for the BTFSC and BTFSS (Fetch) instructions of the present invention;

[0096] Figure 76 is a flowchart for the BTFSC and BTFSS (Execute) instructions of the present invention;

[0097] Figure 77 is a flowchart for the Literal Operations (Fetch) of the present invention;

[0098] Figure 78 is a flowchart for the Literal Operations (Execute) of the present invention;

[0099] Figure 79 is a flowchart for the LFSR (Fetch) instruction of the present invention;

[0100] Figure 80 is a flowchart for the LFSR (Execute1) instruction of the present invention;

[0101] Figure 81 is a flowchart for the LFSR (Execute2) instruction of the present invention;

[0102] Figure 82 is a flowchart for the DAW (Fetch) instruction of the present invention;

[0103] Figure 83 is a flowchart for the DAW (Execute) instruction of the present invention;

[0104] Figure 84 is a flowchart for the MULLW (Fetch) instruction of the present invention;

[0105] Figure 85 is a flowchart for the MULLW (Execute) instruction of the present invention;

[0106] Figure 86 is a flowchart for the CLRWDT, HALT, RESET, and SLEEP (Fetch) instructions of the present invention;

[0107] Figure 87 is a flowchart for the CLRWDT, HALT, RESET, and SLEEP (Execute) instructions of the present invention;

[0108] Figure 88 is a flowchart for the MOVELB (Fetch) instruction of the present invention;

[0109] Figure 89 is a flow chart for the MOVLB (Execute) instruction of the present invention;

[0110] Figure 90 is a flow chart for the Branch Operations (Fetch) of the present invention;

[0111] Figure 91 is a flow chart for the Branch Operations (Execute) of the present invention;

[0112] Figure 92 is a flow chart for BRA and RCALL (Fetch) instructions of the present invention;

[0113] Figure 93 is a flow chart for BRA and RCALL (Execute) instructions of the present invention;

[0114] Figure 94 is a flow chart for PUSH (Fetch) instruction of the present invention;

[0115] Figure 95 is a flow chart for PUSH (Execute) instruction of the present invention;

[0116] Figure 96 is a flow chart for POP (Fetch) instruction of the present invention;

[0117] Figure 97 is a flow chart for POP (Execute) instruction of the present invention;

[0118] Figure 98 is a flow chart for RETURN and RETFIE (Fetch) instructions of the present invention;

[0119] Figure 99 is a flow chart for RETURN and RETFIE (Execute) instructions of the present invention;

[0120] Figure 100 is a flow chart for RETLW (Fetch) instruction of the present invention;

[0121] Figure 101 is a flow chart for RETLW (Execute) instruction of the present invention;

[0122] Figure 102 is a flow chart for GOTO (Fetch) instruction of the present invention;

[0123] Figure 103 is a flow chart for GOTO (Execute1) instruction of the present invention;

[0124] Figure 104 is a flow chart for GOTO (Execute2) instruction of the present invention;

[0125] Figure 105 is a flow chart for CALL (Fetch) instruction of the present invention;

[0126] Figure 106 is a flow chart for CALL (Execute1) instruction of the present invention;

[0127] Figure 107 is a flow chart for CALL (Execute2) instruction of the present invention;

[0128] Figure 108 is a flow chart for TBLRD*, TBLRD*+, TBLRD*-, and TBLRD+* (Fetch) instructions of the present invention;

[0129] Figure 109 is a flow chart for TBLRD*, TBLRD*+, TBLRD*-, and TBLRD+* (Execute1) instructions of the present invention;

[0130] Figure 110 is a flow chart for TBLRD*, TBLRD*+, TBLRD*-, and TBLRD+* (Execute2) instructions of the present invention;

[0131] Figure 111 is a flow chart for TBLWT*, TBLWT*+, TBLWT*-, and TBLWT+* (Fetch) instructions of the present invention;

[0132] Figure 112 is a flow chart for TBLWT*, TBLWT*+, TBLWT*-, and TBLWT+* (Execute) instructions of the present invention;

[0133] Figure 113 is a flow chart for TBLWT*, TBLWT*+, TBLWT*-, and TBLWT+* (Execute2) instructions of the present invention; and

[0134] Figure 114 is an instruction decode map of the present invention.

[0135] Figure 115 is a block diagram illustrating an alternate paging scheme according to the teachings of the present invention.

[0136] Figure 116 is a block diagram illustrating an alternate paging scheme according to the teachings of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0137] The present invention is an apparatus, method and system for providing, in several embodiments, a microcontroller instruction set and microcontroller architecture that includes a linearized address space that enables modular emulation.

[0138] The architecture of the apparatus of the preferred embodiment of the present invention modifies the prior art Harvard architecture in that the data path is 8-bit and the instruction length is 16-bit with a four-phase internal clocking scheme. Moreover, the preferred embodiment has a linearized memory addressing scheme that eliminates the need for paging and banking. The memory addressing scheme of the present invention allows for program memory addressability up to 2M bytes. Emulation of modules is also supported by the present invention.

[0139] The present invention overcomes the above-identified problems as well as other shortcomings and deficiencies of existing technologies by providing a microcontroller instruction set that eliminates many of the compiler errors experienced in the prior art. Moreover, an apparatus and system is provided that enables a linearized address space that makes modular emulation possible.

[0140] The present invention can directly or indirectly address its register files or data memory. All special function registers, including the Program Counter (PC) and Working Register (W), are mapped in the data memory. The present invention has an orthogonal (symmetrical) instruction set that makes it possible to carry out any operation on any register using any addressing mode. This symmetrical nature and lack of 'special optimal situations' make programming with the present invention simple yet efficient. In addition, the learning curve is reduced significantly. One of the present invention family architectural enhancements from the prior art allows two file registers to be used in some two operand instructions. This allows data to be moved directly between two registers without going through the W register and thus increasing performance and decreasing program memory usage. Figure 6 shows a block diagram for the microcontroller core of the present invention.

[0141] The microcontroller core 100 of the present invention is illustrated in Figure 6. By convention, connecting signal lines in Figure 6 can contain a slash with an adjacent number indicating the bandwidth (in bits) of the signal line. Referring to the upper right corner of Figure 6, we find a data memory 104 that is used for storing and transferring data to and from a central processing unit (described below). The data memory 104 is composed of a plurality of address locations. In the preferred embodiment of the present invention, the data memory 104 is a linearized 4K memory which is divided into a plurality of sixteen pages or banks. Typically,

each bank has 256 address locations. In the preferred embodiment, one of the plurality of banks is a dedicated to general and specific purpose registers, in this case the topmost bank, bank 0.

[0142] A selection circuit 108 is coupled to the data memory 104 through an address latch 102. The selection circuit 108 is used for selecting one of the plurality of sources that supply the bank address values in the data memory 104

[0143] The preferred embodiment of the present invention includes an ALU 140 with working (W) register 136, a PLA, an 8-bit multiplier, a program counter (PC) 168 with stack 170, a table latch 124, table pointer 148, a ROM latch 152 with IR latch 126, FSRs 120, 121, 122, interrupt vectoring circuitry, and most common status registers. Unlike the prior art, the design of the present invention obviates the need for a timer in a separate module, all reset generation circuitry (WDT, POR, BOR, etc.), interrupt flags, enable flags, INTCON registers, RCON registers, configuration bits, device ID word, ID locations, and clock drivers.

I/O List:

[0144] A generous list of input/output (I/O) commands are available with the present invention, the I/O list is shown in Table 1.

Table 1 I/O List

Name	Count I/O	Normal Operation	Operation Test Module	Program Module	Emulation Module
addr<21:0>	22/O	Program Memory address			
nqbank<3:0>	4/O	Active low RAM bank selection			
d<15:0>	16/I	Program memory data			
db<7:0>	8/ I/O	Data bus			
forcext	1/I		Force external instruction test mode		
irp<7:0>	8/O	Peripheral Address			
irp9	1/O	Instruction register bit 9			

ncodeprt	1/I	Active low code protect			
neprtim	1/I	Active low end of EPROM write			
nhalt	1/I				Active low halt
nintake	1/I	Active low interrupt acknowledge early and wake up from sleep			
np<7:0>	8/O	Table latch data			
npcmux	1/O	Active low PC mulitplex			
npchold	1/O	Active low PC hold			
nprtchg	1/I	Active low port change interrupt			
nq4clrwdt	1/O	Active low clear wdt			
nq4sleep	1/O	Active low sleep			
nqrd	1/O	Active low read file			
nreset	1/I	Active low reset			
nwrf	1/O	Active low write file			
q1:q4	4/I	4-phase Q clocks			
q13	1/I	Combination of Q clocks			
q23	1/I	Combination of Q clocks			
q41	1/I	Combination of Q clocks			
test0	1/I		Test mode 0		
tsthvdet	1/I	High voltage detect			
wreprom	1/O	Write eprom			
writem	1/O	Write memory			
wrtbl	1/O	Table write instruction			
nintakd	1/I	Interrupt acknowledge delayed			
intak	1/I	Interrupt acknowledge			

Clocking Scheme/Instruction Cycle

[0145] The clock input (from OSC1) is internally divided by four to generate four non-overlapping quadrature clocks, namely Q1, Q2, Q3, and Q4 as shown in Figure 7. Internally, the program counter (PC) is incremented every Q1, and the instruction is fetched from the program memory and latched into the instruction register using Q4. The instruction is decoded and

executed during the following Q1 through Q4. The PLA decoding is done during Q1. During the Q2 and Q3 cycle, the operand is read from memory or peripherals and the ALU performs the computation. During Q4 the results are written to the destination location. The clocks and instruction execution flow are shown in Figure 8.

Q Cycle Activity

[0146] Each instruction cycle (TCY) is comprised of four Q cycles (Q1-Q4) as shown in Figure 7. The Q cycle is the same as the device oscillator cycle (TOSC). The Q cycles provide the timing/designation for the Decode, Read, Process Data, Write etc., of each instruction cycle. The following diagram (Figure 7) shows the relationship of the Q cycles to the instruction cycle. The four Q cycles that make up an execution instruction cycle (TCY) can be generalized as:

Q1: Instruction Decode Cycle or forced NOP
Q2: Instruction Read Cycle or NOP
Q3: Process the Data
Q4: Instruction Write Cycle or NOP

[0147] Each instruction will show the detailed Q cycle operation for the instruction.

Instruction Flow/Pipelining

[0148] An "Instruction Cycle" consists of four Q cycles (Q1, Q2, Q3, and Q4). The instruction fetch and execute are pipelined such that fetch takes one instruction cycle while decode and execute takes another instruction cycle. However, due to the pipelining, each instruction effectively executes in one cycle. There are four types of instruction flows. First is a normal 1-word 1 cycle pipelined instruction. These instructions will take one effective cycle to execute as shown in Figure 9. Second is a 1 word 2 cycle pipeline flush instruction. These instructions include the relative branches, relative call, skips and returns. When an instruction changes the PC, the pipelined fetch is discarded. This makes the instruction take two effective cycles to execute as shown in Figure 10. Third are the table operation instructions. These

instructions will suspend the fetching to insert and read or write cycle to the program memory. The instruction fetched while executing the table operation is saved for 1 cycle and executed in the cycle immediately after the table operation as shown in Figure 11. Fourth are new two word instructions. These instructions include MOVFF and MOVLF. In these instructions, the fetch after the instruction contains the remainder of the addresses. For a MOVFF instruction during execution of the first word, the machine will execute a read of the source register. During execution of the second word, the source address is obtained, and then the instruction will complete the move as shown in Figure 12. The MOVLF is similar although it moves 2 literal values into FSRnH and FSRnL in 2 cycles as shown in Figure 13. Fifth, is the two word instructions for CALL and GOTO. In these instructions, the fetch after the instruction contains the remainder of the jump or call destination addresses. Normally, these instructions would require 3 cycles to execute, 2 for fetching the 2 instruction words and 1 for the subsequent pipeline flush. However, by providing a high-speed path on the second fetch, the PC can be updated with the complete value in the first cycle of instruction execution, resulting in a 2 cycle instruction as shown in Figure 14. Sixth, is the interrupt recognition execution. Instruction cycles during interrupts are discussed in the interrupts section below.

The ALU

[0149] The present invention contains an 8-bit Arithmetic and Logic Unit (ALU)142 and working register 136 as shown in Figure 6. The ALU 142 is a general purpose arithmetic unit. It performs arithmetic and Boolean functions between data in the working register and any register file. The ALU 142 is 8-bits wide and capable of addition, subtraction, shift, and logical operations. Unless otherwise mentioned, arithmetic operations are two's complement in nature. The working (W) register 136 is an 8-bit working register used for ALU 140 operations. The W

register 136 is addressable and can be directly written or read. The ALU 140 is capable of carrying out arithmetic or logical operations on two operands or a single operand. All single operand instructions operate either on the W register 136 or the given file register. For two operand instructions, one of the operands is the W register 136 and the other one is either a file register or an 8-bit immediate constant, or an equivalent storage medium.

[0150] Depending on the instruction executed, the ALU 140 may affect the values of the Carry (C), Digit Carry (DC), Zero (Z), Overflow (OV), and Negative (N) bits in the STATUS register (discussed below). The C and DC bits operate as a borrow and digit borrow out bit, respectively, in subtraction.

[0151] The preferred embodiment of the present invention includes an 8 x 8 hardware multiplier 134 included in the ALU 142 of the device as shown in Figure 6. By making the multiply a hardware operation, the operation completes in a single instruction cycle. This hardware operation is an unsigned multiply that gives a 16-bit result. The result is stored into the 16-bit product register (PRODH:PRODL). The multiplier does not affect any flags in the STATUS register.

Status Registers

[0152] The STATUS register contains the status bits of the ALU 140. The status register is shown in Figure 15. In the preferred embodiment of the present invention, bit 7-5 are unimplemented and are read as '0'.

[0153] bit 4 is "N", the Negative bit. This bit is used for signed arithmetic (2's complement). It indicates whether the result was negative, (ALU MSb = 1), 1 = Result was negative, 0 = Result was positive.

[0154] bit 3 is the "OV" Overflow bit. This bit is used for signed arithmetic (2's complement). It indicates an overflow of the 7-bit magnitude, which causes the sign bit (bit 7) to

change state. For this bit, 1 = Overflow occurred for signed arithmetic, (in this arithmetic operation), and 0 = No overflow occurred.

[0155] bit 2 is the "Z" Zero bit. For this bit, 1 = The result of an arithmetic or logic operation is zero, and 0 = The results of an arithmetic or logic operation is non-zero.

[0156] bit 1 is the "DC" Digit carry/borrow bit. For this bit, 1 = A carry-out from the 4th low order bit of the result occurred, and 0 = No carry-out from the 4th low order bit of the result It should be noted that, for borrow, the polarity is reversed.

[0157] bit 0 is the "C" carry/borrow bit. For this bit, 1 = A carry-out from the most significant bit of the result occurred, and 0 = No carry-out from the most significant bit of the result. As with bit 1, for borrow the polarity is reversed.

[0158] The C and DC bits operate as a borrow and digit borrow bit, respectively, in subtraction. Carry is ALU bit 7 carry out. Digit Carry is ALU bit 3 carry out. Zero is true if ALU result bit <7:0> is '0'. N is ALU result bit 7. The overflow bit will be set if the 2's complement result exceeds +127 or is less than -128. Overflow is ALU bit 6 carry out XOR ALU bit 7 carry out. As with all the other registers, the STATUS register can be the destination for any instruction. If the STATUS register is the write destination for an instruction that affects any of the status bits, then the write to the status bits is disabled. The bits are set or cleared according to the ALU results and the instruction specification. Therefore, the result of an instruction with the STATUS register as destination may be different than intended.

[0159] For example, the CLRF REG instruction normally writes the register to 0 and sets the Z bit. The CLRF STATUS instruction will disable the write to the N, OV, DC and C bits and set the Z bit. This leaves the STATUS register as 000u u1uu. It is recommended, therefore, that only the BCF, BSF, SWAPF and MOVWF instructions be used to alter the STATUS register

because these instructions do not affect any status bit. To see how other instructions affect the status bits, see the "Instruction Set Summary."

Program Counter Module

[0160] The program counter (PC)168 (see Figure 6) is modified to allow expansion up to a maximum of 21 bits. This is done by adding a 5-bit wide PCLATU register that operates similarly to the PCLATH register. The PC 168 is also modified to address bytes rather than words in the program memory. To implement this, there is a byte addressing bit at the LSb of the PC 168 that is always 0. The LSb bit of the PCL is readable but not writeable. Should the user attempt to write a '1' into the LSb, the result will be a '0'. To allow hidden test EPROM, there is a hidden 22nd bit (bit21) of PC 168 (see Figure 16). This PC bit is normally 0. When entering test mode or programming mode, this bit is set and the instructions will be fetched from the test area. Once this bit is set, it cannot be cleared by program execution, the device must be reset.

[0161] The Program Counter (PC) 168 is up to a 21-bit register as shown in Figure 16. PCL 184, the low byte of the PC 168, is mapped in the data memory 104 (see Figure 6). PCL 184 is readable and writeable just as is any other register. PCH 182 and PCU 180 are the high bytes of the PC and are not directly addressable. Since PCH 182 and PCU 184 are not mapped in data or program memory 160, registers PCLATH 178 (PC high latch) and PCLATU 176 (PC upper latch) are used as holding latches for the high bytes of the PC 168.

[0162] PCLATH 178 and PCLATU 176 are mapped into data memory 104. The user can read and write PCH 182 through PCLATH 178 and PCU 180 through PCLATU 176. The PC 168 is word incremented by 2 after each instruction fetch during Q1 unless:

- Modified by a `GOTO`, `CALL`, `RETURN`, `RETLW`, `RETFIE`, or Branch instruction.

- Modified by an interrupt response.
- Due to destination write to PCL 168 by an instruction.

"Skips" are equivalent to a forced NOP cycle at the skipped address. Figures 16 and 17 show the operation of the program counter for various situations.

[0163] Referring to Figure 16, the operations of the PC 168, PCLATH 178, and PCLATU 176 for different instructions are as follows:

a. Read instructions on PCL:

[0164] For any instruction that reads PCL 184. All byte instructions with d=0; `MOVFF PCL, X;` CPFSEQ; CPFSGT; CPFSLT; MULWF; TSTFSZ then PCL to data bus then to ALU or to the destination. Finally, PCH to PCLATH and PCU to PCLATU.

b. Write instructions on PCL:

[0165] Any instruction that writes to PCL 184. For example, MOVWF; CLRF; SETF, then write 8-bit data to the data bus 174 and then to PCL 184. Also, PCLATH to PCH, and PCLATU to PCU.

c. Read-Modify-Write instructions on PCL:

[0166] Any instruction that does a read-write-modify operation on PCL. All byte instructions with d=1; Bit Instructions; NEGF. Read: PCL to data bus to ALU. Write: write the 8-bit result to data bus and to PCL; then PCLATH to PCH; and finally PCLATU to PCU.

[0167] The read-modify-write only affects the PCL 184 with the result. PCH 182 and PCU 180 are loaded with the value in the PCLATH 178 and PCLATU 176 respectively. For example, for the instruction "ADDWF", PCL 184 will result in the following jump. If PC = 0003F0h, W = 30h, PCLATH = 05h and PCLATU = 1h before the instruction, PC = 010520h after the instruction. To accomplish a true 20-bit computed jump, the user needs to compute the

20-bit destination address, write to PCLATH 178 and PCLATU 176, and then write the low value to PCL 168.

d. RETURN instruction:

[0168] Stack<MRU> to PC<20:0> Using Figure 17, the operation of the PC 168 , PCLATH 178, and PCLATU 176 for the GOTO and the CALL instructions is as follows:

e. CALL, GOTO instructions:

[0169] A destination address is provided in the 2-word instruction (opcode). The first Word Opcode<6:0> to PCL<7:1>. The first Word Opcode<7> to PCLATH<0> and to PCH<0>. The second Word Opcode<6:0> to PCLATH<7:1> and PCH <7:1>. The second Word Opcode<11:7> to PCLATU<4:0> and PCU <4:0>.

[0170] It should be noted that the following PC 168 related operations do not change PCLATH 178 and PCLATU 176:

a. RETLW, RETURN, and RETFIE instructions.
b. Interrupt vector is forced onto the PC.
c. Read-modify-write instructions on PCL (e.g. BSF PCL, 2).

Return Stack Operation

[0171] The present invention has a 31 level deep return (or hardware) stack. The depth of the stack was increased over the prior art in order to allow more complex programs. The stack is not part of either the program or data memory space.

[0172] The PC 168 is pushed onto the stack when a CALL or RCALL instruction is executed or an interrupt is acknowledged. The PC 168 value is pulled off the stack on a RETURN, RETLW, or a RETFIE instruction. PCLATU 176 and PCLATH 178 are not affected by any of the return instructions.

[0173] The stack operates as a 31 word by 21 bit RAM and a 5-bit stack pointer, with the stack pointer initialized to 00000b after all resets. There is no RAM word associated with stack

pointer 000h. This is only a reset value. During a CALL type instruction causing a push onto the stack, the stack pointer is first incremented and the RAM location pointed to by the stack pointer is written with the contents of the PC. During a RETURN type instruction causing a pop from the stack, the contents of the RAM location pointed to by the STKPTR is transferred to the PC and then the stack pointer is decremented.

Top Of Stack Access

[0174] The top of the stack is readable and writeable. Three register locations, TOSU, TOSH and TOSL address the stack RAM location pointed to by the STKPTR. This allows users to implement a software stack if necessary. After a CALL or RCALL instruction or an interrupt, the software can read the pushed value by reading the TOSU, TOSH and TOSL registers. These values can be placed on a user defined software stack. At return time, the software can replace the TOSU, TOSH and TOSL and do a return. It should be noted that the user must disable the global interrupt enable bits during this time to prevent inadvertent stack operations.

PUSH and POP instructions

[0175] Since the Top-of-stack (TOS) is readable and writeable, the ability to push values onto the stack and pull values off the stack without disturbing normal program execution is a desirable option. To push the current PC value onto the stack, a PUSH instruction can be executed. This will push the current PC value onto the stack; setting the TOS = PC and PC = PC + 2. The ability to pull the TOS value off of the stack and replace it with the value that was previously pushed onto the stack, without disturbing normal execution, is achieved by using the POP instruction. The POP instruction pulls the TOS value off the stack, but this value is not written to the PC; the previous value pushed onto the stack then becomes the TOS value.

Return Stack Pointer (STKPTR)

[0176] The STKPTR register contains the return stack pointer value and the overflow and underflow bits. The stack overflow bit (STKOVF) and underflow bit (STKUNF) allow software verification of a stack condition. The STKOVF and STKUNF bits are cleared after a POR reset only.

[0177] After the PC is pushed onto the stack 31 times (without popping any values off the stack), the 32nd push over-writes the value from the 31st push and sets the STK-OVF bit while the STKPTR remains at 11111b. The 33rd push overwrites the 32nd push (and so on) while STKPTR remains 11111b.

[0178] After the stack is popped enough times to unload the stack, the next pop will return a value of zero to the PC and sets the STKUNF bit while the STKPTR remains at 00000b. The next pop returns zero again (and so on) while STKPTR remains 00000b. Note that returning a zero to the PC on an underflow has the effect of vectoring the program to the reset vector where the stack conditions can be verified and appropriate actions can be taken.

[0179] The stack pointer can be accessed through the STKPTR register. The user may read and write the stack pointer values. This can be used by RTOS for return stack maintenance. Figure 18 shows the STKPTR register. The value of the stack pointer will be 0 through 31. At reset the stack pointer value will be 0. The stack pointer when pushing will increment and when popping will decrement.

Stack Overflow/Underflow Resets

[0180] At the user's option, the overflow and underflow can cause a device reset to interrupt the program code. The reset is enabled with a configuration bit, STVRE. When the STVRE bit is disabled, an overflow or underflow will set the appropriate STKOVF or STKUNF bit and not cause a reset. When the STVRE bit is enabled, a over-flow or underflow will set the

appropriate STKOVF or STKUNF bit and then cause a device reset very similar in nature to the WDT reset. In either case, the STKOVF or STKUNF bits are not cleared unless the user software clears them or a POR reset clears them. Figures 18-21 illustrate stack registers. Figures 22-29 illustrate stack operations.

Program Memory

[0181] The preferred embodiment of the present invention has up to a 2 Megabyte (2M) x 8 user program memory space. The program memory space is primarily to contain instructions for execution, however, data tables may be stored and accessed using the table read and write instructions. Another 2M x 8 test program memory space is available for test ROM, configuration bits, and identification words.

[0182] The devices have up to a 21-bit program counter capable of addressing the 2M x 8 program memory space. There is also a 22nd PC bit that is hidden during normal operation, and when it is set, it is possible to access configuration bits, device ID and test ROM. This bit can be set in test mode or programming mode, and the device must be reset to clear this bit. User program memory space cannot be accessed with this bit set. Because the PC must access the instructions in program memory on an even byte boundary, the LSb of the PC is an implied '0' and the PC increments by two for each instruction.

[0183] The reset vector is at 000000h and the high priority interrupt vector is at 000008h and the low priority interrupt vector is at 000018h (see Figure 30).

Program Memory Organization

[0184] Each location in the program memory has a byte address. In addition, each 2 adjacent bytes have a word address. Figure 31 shows the map of the program memory with byte and word addresses shown. Within the program memory, the instructions must be word aligned. Figure 32 shows the map of the program memory with several example instructions and the hex

codes for those instructions placed into the map. Table operations will work with byte entities. A table block is not required to be word aligned, so a table block can start and end at any byte address. The exception to this is if a table write is being used to program the internal program memory or an external word wide flash memory. When programming, the write data may need to be aligned to the word width used by the programming method.

Program Memory Modes

[0185] The present invention can operate in one of five possible program memory configurations. The configuration is selected by configuration bits. The possible modes are:

- MP - Microprocessor
- EMC - Extended Microcontroller
- PEMC - Protected Extended Microcontroller
- MC - Microcontroller
- PMC - Protected Microcontroller

[0186] The microcontroller and protected microcontroller modes only allow internal execution. Any access beyond the program memory reads all zeros. The protected micro-controller mode also enables the code protection feature. Microcontroller is the default mode of an un-programmed device.

[0187] The extended microcontroller mode accesses both the internal program memory as well as external program memory. Execution automatically switches between internal and external memory. The 21-bits of address allow a program memory range of 2M-bytes. The protected extended microcontroller mode will code protect the internal program memory by preventing table reads/writes to the internal memory while still allowing execution and table reads/writes of the external program memory.

[0188] The microprocessor mode only accesses the external program memory. The on-chip program memory is ignored. The 21-bits of address allow a program memory range of 2M-bytes.

[0189] Test memory and configuration bits are readable during normal operation of the device by using the TBLRD instruction. These areas are only modifiable using the TBLWT instruction if the LWRT bit in the RCON register is set or the device is in test and programming mode.

[0190] These areas can only be executed from in test and programming mode.

[0191] The extended microcontroller mode and microprocessor modes are available only on devices which have the external memory bus defined as part of the I/O pins. Table 2 lists which modes can access internal and external memory. Figure 33 illustrates the device memory map in the different program modes.

Table 2 Device Mode Memory Access

Operating Mode	Internal Program Memory	External Program Memory
Microprocessor	No Access	Execution / TBLRD / TBLWT
Extended Microcontroller	Execution / TBLRD / TBLWT	Execution / TBLRD / TBLWT
Protected Extended Microcontroller	Execution	Execution / TBLRD / TBLWT
Microcontroller	Execution / TBLRD / TBLWT	No Access
Protected Microcontroller	Execution / TBLRD	No Access

External Program Memory Interface

[0192] When either microprocessor or extended microcontroller mode is selected, up to four ports are configured as the system bus. Two ports and part of a third are the multiplexed

address/data bus and part of one other port is used for the control signals. External components are needed to demultiplex the address and data. The external memory interface can run in 8-bit data mode or 16-bit data mode. Addresses on the external memory interface are byte addresses always.

[0193] Figures 36 and 37 describe the external memory connections for 16-bit and 8-bit data respectively. The external program memory bus shares I/O port functions on the pins. Figure 38 lists a typical mapping of external bus functions on I/O pin functions. In extended microcontroller mode, when the device is executing out of internal memory, the control signals

[0194] will NOT be active. They will go to a state where the AD<15:0>, A<19:0> are tri-state; the OE, WRH, WRL, UB and LB signals are '1'; UBA0 and ALE is '0'.

16-Bit External Interface

[0195] If the external interface is 16-bit, the instructions will be fetched as 16-bit words. The OE output enable signal will enable both bytes of program memory at once to output a 16-bit word. The least significant bit of the address, BA0, need not be connected to the memory devices.

[0196] An external table read is logically performed one byte at a time, although the memory will read a 16-bit word externally. The least significant bit of the address will internally select between high and low bytes (LSb = 0 to lower byte, LSb = 1 to upper byte). The external address in microprocessor and extended microcontroller modes is 21-bits wide; this allows addressing of up to 2M-bytes.

[0197] An external table write on a 16-bit bus is logically performed one byte at a time. The actual write will depend on the type of external device connected and the WM<1:0> bits in the MEMCON register, shown in Figure 34. The Table Operations section details the actual write cycles.

8-Bit External Interface

[0198] If the external interface is 8-bit, the instructions will be fetched as 2 8-bit bytes. The two bytes are fetched within one instruction cycle. The least significant bit of the address must be connected to the memory devices. The OE output enable signal and BA0=1 will enable the most significant byte of the instruction to read from program memory for the Q3 portion of the cycle, then BA0 will change to 0 and the least significant byte will be read for the Q4 portion of the cycle; to form the 16-bit instruction word.

[0199] An external table read is also performed one byte at a time. An external table write is performed one byte at a time. The WRL is active on every external write.

[0200] When 8-bit interface is selected, the WRH, UB and UL lines are not used and the pins revert to I/O port functions. A configuration bit selects the 8-bit mode of the external interface.

External Wait Cycles

[0201] The external memory interface supports wait cycles. The external memory wait cycles only apply to the table read and table write operations over the external bus. Since the device execution is tied to instruction fetches, there is no sense to execute faster than the fetch rate. So if the program fetches need to be slowed, the processor speed must be slowed with a different TCY time.

[0202] The WAIT <1:0> bits in the MEMCON register will select 0, 1, 2 or 3 extra TCY cycles per memory fetch cycle. The wait cycles will be effective for table reads and writes on a 16-bit interface. On an 8-bit interface, for table reads and writes, the wait will only occur on the Q4.

[0203] The default setting of the wait on power up is to assert a wait of the maximum of the 3 TCY cycles. This insures that slow memories will work in microprocessor mode

immediately after reset. A configuration bit, called WAIT, will enable or disable the wait states. Figure 39 illustrates the 16-bit interface and Figure 40 illustrates the 8-bit, in both cases showing program memory instruction fetches with no waits and table reads with wait states.

External Bus Signal Disables

[0204] To allow flexibility in the utilization of the pins committed to the external bus, several disables are provided in configuration bits. Also, to disable the entire external bus, as might be done while in extended microcontroller mode and allowing a DMA function, the EBDIS bit in the MEM-CON, shown in Figure 35, register. This disable will allow the user to tri-state the entire external bus inter-face. This will allow DMA operations as well as direct control of external devices by program control through the I/O pin functions.

[0205] In emulator systems, the -ME devices must have inputs to represent the bus disable configuration bits to allow the I/O port functions to detect the status of the pins as external interface. The -ME device also has a special input pin that indicates if the emulator system is in the microprocessor or extended microcontroller mode.

Data Memory

[0206] The data memory and general purpose RAM size can be extended to 4096 bytes in the present invention. The data memory address is 12-bits wide. The data memory is partitioned into 16 banks of 256 bytes which contain the General Purpose Registers (GPRs) and Special Function Registers (SFRs).

[0207] The GPR's are mechanized into a byte wide RAM array of the size of the combined GPR registers. The SFR's are typically distributed among the peripherals whose functions they control.

[0208] The bank is selected by the bank select register (BSR<3:0>). The BSR register can potentially access more than 16 banks, however the direct long addressing mode is limited to 12-bit addresses or 16 banks. The BSR is limited accordingly.

[0209] Device instructions can read, modify and write a particular location in one instruction cycle. There is only one address generation per cycle, so it is not possible to read one location and modify/write another in a single cycle. Figure 42 shows an example data memory map.

General Purpose Registers

[0210] In all PIC devices, all data RAM is available for use as registers by all instructions. Most banks of data memory only contain GPR memory. There must be GPR memory included in bank 0 on all devices.

[0211] The absolute minimum for the number of GPRs in bank 0 is 128. This GPR area, called the Access RAM, is essential for allowing programmers to have a data structure that is accessible regardless of the setting of the BSR.

Special Function Registers

[0212] SFR are special registers, typically used for device and peripheral control and status functions. They are accessible by all instructions. All SFRs should be contained in the upper 128 bytes of bank 15, if possible. If the SFRs do not use all the available locations on a particular device, the unused locations will be unimplemented and read as '0's. Certain devices, such as LCD controllers may have SFR areas in other banks than bank 15.

[0213] The boundary of the SFR's in bank 15 can be modified from device to device. At least 16 GPR's must be included in the Access Bank. Figure 43 displays a possible Special

Function Register map. Figures 44 and 45 displays a summary of the core Special Function Registers.

Addressing Modes

[0214] There are 7 data addressing modes supported by the present invention:

- inherent
- literal
- direct short
- direct forced
- direct long
- indirect
- indexed indirect offset

Three of the modes, direct forced, direct long and indirect indexed, are new to the PIC architecture.

Inherent

[0215] Some instructions such as DAW do not require addressing other than that explicitly defined in the opcode.

Literal

[0216] Literal instructions contain a literal constant field, typically used in a mathematical operation such as ADDLW. Literal addressing is also used for GOTO, CALL, and branch opcodes.

Direct Short

[0217] Most mathematical and move instructions operate in the direct short addressing mode. In this addressing mode, the instruction contains eight bits of least significant address for the data. The remaining four bits of address are from the Bank Select Register or BSR. The BSR is used to switch between banks in the data memory area (see Figure 47).

[0218] The need for a large general purpose memory space dictated a general purpose RAM banking scheme. The lower nibble of the BSR selects the currently active general purpose RAM bank. To assist this, a MOVLB bank instruction has been provided in the instruction set.

[0219] If the currently selected bank is not implemented (such as Bank 13), any read will read all '0's. Any write is completed to the bit bucket and the STATUS register bits will be set/cleared as appropriate.

Direct Forced

[0220] All the Special Function Registers (SFRs) are mapped into the data memory space. In order to allow easy access to the SFR's, they are all, generally, mapped in Bank 15. To simplify access, there is a 1 bit field in the instruction that points the address to the lower half of bank 0 for common RAM and the upper half of bank 15 for the SFR's regardless of the contents of the BSR. With the BSR set to BSR=n then, it is possible to address 3 banks with any instruction; Bank 0 and 15 in direct forced mode and Bank "n" in direct short mode.

Direct Long

[0221] The direct long addressing codes all twelve bits of the data address into the instruction. Only the MOVFF instruction uses this mode.

Indirect Addressing

[0222] Indirect addressing is a mode of addressing data memory where the data memory address in the instruction is determined by another register. This can be useful for data tables or stacks in the data memory. Figure 53 shows the operation of indirect addressing. The value of the FSR register is used as the data memory address.

Indirect Addressing Registers

[0223] The present invention has three 12-bit registers for indirect addressing. These registers are:

- FSR0H and FSR0L
- FSR1H and FSR1L
- FSR2H and FSR2L

The FSR's are 12-bit registers and allow addressing anywhere in the 4096-byte data memory address range.

[0224] In addition, there are registers INDF0, INDF1 and INDF2 which are not physically implemented. Reading or writing to these registers activates indirect addressing, with the value in the corresponding FSR register being the address of the data. If file INDF0 (or INDF1,2) itself is read indirectly via an FSR, all '0's are read (Zero bit is set). Similarly, if INDF0 (or INDF1,2) is written to indirectly, the operation will be equivalent to a NOP, and the STATUS bits are not affected.

Indirect Addressing Operation

[0225] Each INDF register has four addresses associated with it. When a data access is done to the one of the four INDF locations, the address selected will configure the FSR register to:

- Auto-decrement the value (address) in the FSR after an indirect access (post-decrement)
- Auto-increment the value (address) in the FSR after an indirect access (post-increment)
- Auto-increment the value (address) in the FSR before an indirect access (pre-increment)
- No change to the value (address) in the FSR after an indirect access (no change).

When using the auto-increment or auto-decrement features, the effect on the FSR is not reflected in the STATUS register. For example, if the indirect address causes the FSR to equal '0', the Z bit will not be set. Adding these features allows the FSR to be used as a stack pointer in addition to its uses for data table operations.

Indexed Indirect Addressing

[0226] Each INDF has an address associated with it that performs an indexed indirect access. When a data access to this INDF location occurs, the FSR is configured to:

- Add the signed value in the W register and the value in FSR to form the address before an indirect access.
- The FSR value is not changed.

Indirect Writing of Indirect Addressing (INDF) Registers

[0227] If an FSR register contains a value that points to one of the indirecting registers (FEFh-FEBh, FE7h-FE3h, FDFh-FDBh), an indirect read will read 00h (Zero bit is set) while an indirect write will be equivalent to a NOP (STATUS bits are not affected).

Indirect Writing of Pointer (FSR) Registers

[0228] If an indirect addressing operation is done where the target address is an FSRnH or FSRnL register, the write operation will dominate over the pre or post increment/decrement functions. For example:

FSR0 = FE8h (one less than the location of FSR0L)

W=50h

MOVWF *(++FSR0) ;(PREINC0)

will increment FSR0 by one to FE9h, pointing to FSR0L. Then write of W into FSR0L will change FSR0L to 50h. However,

FSR0 = FE9h (the location of FSR0L)

W = 50h

MOVWF *FSR0++ ;(POSTINC0)

will attempt to write W into the FSR0L at the same time the increment of FSR0 is to occur. The write of W will prevail over the post increment and FSR0L will be 50h.

Instruction Set Summary

[0229] The instruction set of the present invention consists of 77 instructions. Due to excessive page and bank switching in prior art architectures, the Program and Data memory maps needed to be linearized, and the instruction set was modified to facilitate this linearization. The Data Memory space of the preferred embodiment of the present invention has a maximum of 4K bytes, which is made up of 16 banks of 256 bytes each. In the preferred embodiment of the present invention, with all Special Function Registers located in one bank, it is preferred to designate a bit in the opcode of all the instructions that perform file manipulation that could force a virtual bank. Therefore, it is not necessary to switch banks in order to access Special Function Registers.

[0230] The Program Memory space was modified over the prior art systems to be a maximum of 2M bytes in the preferred embodiment. The PC was increased from 13 bits to up to 21 bits, and some instructions that cause a jump (CALL, GOTO) were changed to two-word instructions to load the 21-bit value for the PC. Another improvement over the prior art was the inclusion of a modular emulator. This requires communication between two chips for emulation, and to achieve the desired speeds, it is not possible to have different source and destination registers within the same instruction cycle. Therefore, the MOVPF and MOVFP instructions in the prior art were eliminated. To keep this functionality, a two-word instruction, MOVFF, was added.

[0231] The instruction set of the present invention can be grouped into three types:

- byte-oriented
- bit-oriented
- literal and control operations.

These formats are shown in Figure 56. Figure 54 shows the field descriptions for the opcodes. These descriptions are useful for understanding the opcodes in Figures 57-59 and in each specific instruction description found in Appendix A. Figure 114 shows the instruction decode map.

[0232] For byte-oriented instructions, 'f' represents a file register designator and 'd' represents a destination designator. The file register designator specifies which file register is to be used by the instruction. The destination designator specifies where the result of the operation is to be placed. If 'd' = '0', the result is placed in the W register. If 'd' = '1', the result is placed in the file register specified by the instruction.

[0233] Again, for byte-oriented instructions, 'a' represents the virtual bank select bit. If 'a' = '0', the BSR is overridden and virtual bank is selected. If 'a' = '1', the bank select register (BSR) is not overridden.

[0234] For bit-oriented instructions, 'b' represents a bit field designator which selects the number of the bit affected by the operation, while 'f' represents the address of the file in which the bit is located.

[0235] For literal and control operations, 'k' represents an 8-, 12-, 16- or 20-bit constant or literal value. Moreover, 's' represents the fast call/return select bit. If 's' = '0', the shadow registers are unused. If 's' = '1', the W, BSR and STATUS registers are updated from shadow registers on a RETURN or RETFIE instruction, or the shadow registers are loaded from their corresponding register on a CALL instruction. Finally, 'n' is a 2's complement number that determines the direction and magnitude of the jump for relative branch instructions.

[0236] The instruction set is highly orthogonal and is grouped into:

- byte-oriented operations
- bit-oriented operations
- literal and control operations

All instructions are executed within one single instruction cycle, unless:

- a conditional test is true
- the program counter is changed as a result of an instruction
- a file to file transfer is executed
- a table read or a table write instruction is executed

which in that case, the execution takes two instruction cycles with the second cycle executed as a NOP.

Special Function Registers as Source/Destination

[0237] The orthogonal instruction set of the present invention allows read and write of all file registers, including special function registers. There are some special situations the user should be aware of:

STATUS as destination

[0238] If an instruction writes to the STATUS register, the Z, C, DC, OV and N bits may be set or cleared as a result of the instruction and overwrite the original data bits written.

PCL as source or destination

[0239] Read, write or read-modify-write on PCL may have the following results:

- For a Read PCL, first PCU to PCLATU; then PCH to PCLATH; and then PCL to dest.
- For a Write PCL, first PCLATU to PCU; then PCLATH to PCH; and then 8-bit result value to PCL.
- For a Read-Modify-Write: first PCL to ALU operand, then PCLATH to PCH, then PCLATU to PCU, and then 8-bit result to the PCL.

Where:

PCL = program counter low byte

PCH = program counter high byte

PCLATH = program counter high holding latch

PCU = program counter upper byte

PCLATU = program counter upper holding latch

dest = destination, W or f.

Bit Manipulation

[0240] All bit manipulation instructions are done by first reading the entire register, operating on the selected bit and writing the result back (read-modify-write (R-M-W)). The user should keep this in mind when operating on some special function registers, such as ports. It should be noted that the Status bits that are manipulated by the device (including the Interrupt flag bits) are set or cleared in the Q1 cycle. So there is no issue on doing R-M-W instructions on registers which contain these bits.

[0241] Figures 60-113 contain flowcharts for the general operation of each of the instructions within the instruction set of the present invention. The various Figures show generalized as well as specific steps for the fetching and the execution of the instructions within the instruction set of the present invention. For example, Figure 60 shows the steps for the fetching of byte oriented file register operations, which includes the instructions ADDWF, ADDWFC, ANDWF, COMF, DECF, INCF, IORWF, MOVF, RLCF, RLNCF, RRCF, RRNCF, SUBFWB, SUBWF, SUBWFB, SWAPF, XORWF, MOVWF, and NOP. Similarly, Figure 61 shows the steps for the execution of the byte oriented file register operations, which includes the instructions ADDWF, ADDWFC, ANDWF, COMF, DECF, INCF, IORWF, MOVF, RLCF, RLNCF, RRCF, RRNCF, SUBFWB, SUBWF, SUBWFB, SWAPF, and XORWF (but MOVWF does only a dummy read and NOP does a dummy read and a dummy write).

[0242] Figure 77 shows the fetch steps for the Literal Operations, which includes the instructions: ADDLW, ANDLW, IORLW, MOVLW, SUBLW, and XORLW. As before, Figure 78 shows the execution steps for the Literal Operations, which includes the instructions: ADDLW, ANDLW, IORLW, MOVLW, SUBLW, and XORLW.

[0243] Figure 90 shows a flow chart for the fetching of the Branch Operations, which includes the instructions: BC, BN, BNC, BNN, BNV, BNZ, BV, and BZ. Similarly, Figure 90 shows a flow chart for the execution of the Branch Operations, which includes the instructions: BC, BN, BNC, BNN, BNV, BNZ, BV, and BZ. The remaining figures show the steps of fetching and execution of the other instructions within the instruction set.

[0244] For those multi-Word instructions that require two fetches to obtain the complete instruction, three flowcharts are used to describe the entire fetch and execute process. For example, the MOVFF instruction is described in Figures 70-72. Figure 70 shows a relatively standard fetch operation. However, Figure 71 shows the execution of the first portion of the MOVFF in the left side of the operation boxes while the right portion of the operation boxes show the fetching of the second Word of the instruction. Correspondingly, Figure 72 shows simply the execution steps of the second Word of the MOVFF instruction. Similar flow charts are provided for the other multi-Word instructions: LFSR (Figures 79-81); GOTO (Figures 102-104); CALL (Figures 105-107), TBLRD*, TBLRD*+, TBLRD*-, and TBLRD+* (Figures 108-110); TBLWT*, TBLWT*+, TBLWT*-, and TBLWT+* (Figures 111-113).

[0245] Appendix A contains a detailed listing of the opcodes and instructions of the instruction set of the present invention. The material in Appendix A is incorporated herein by reference for all purposes.

Indexed With Literal Offset

[0246] Figure 115 illustrates another addressing mode, specifically an indexed with literal offset addressing mode. In one embodiment, the indexed with literal offset addressing mode is enabled by programming a context bit called an index bit to a '1'. The index bit will probably be implemented as a fuse, but may also be implemented in software or any flag/switch enabling technique. When the index bit is programmed to be enabled, indexed address with literal offset mode will be address dependent and will also depend upon the value of the Access bit in the instruction word. This mode will only apply to instructions that use direct forced addresses.

[0247] If the Access bit is set to '1', then there is no change in how the address is determined from the previous architecture, and the addressing mode defaults to direct short. If the value of the Access bit is '0', the address contained in the instruction word is decoded and compared to the value 5Fh. If the address is greater than 5Fh, the addressing mode is decoded as direct forced. If the access bit is zero and the address in the instruction word is less than or equal to 05Fh, then addressing mode is indexed with literal offset. When this occurs, the address in the instruction word defaults to a literal value that is added to the contents of FSR2. The resulting value is then used as an address that is to be operated upon.

[0248] Figure 116 shows how the access bank is partitioned when indexed with literal offset enabled. Locations 00h to 5Fh can be mapped to any location in memory. The starting address for this portion of the access bank is mapped to the address contained in the FSR2H:FSR2L registers.

Modifications to FSR2

[0249] In order to support index with literal offset mode, the lower seven bits of the instruction register are included as one of four possible values to add to the contents of FSR2.

The data contained in the IR is considered an unsigned integer, and the result is NOT stored in FSR2. There are four addressing modes:

- Indirect (INDF);
- Indirect with increment/decrement (FSR2+), (+FSR2), and (FSR2-);
- Indirect with offset (FSR2 + W, where the contents of W are used for the offset); and
- Indirect with literal offset (FSR2 + literal).

[0250] The present invention includes a set of instructions that, when invoked, perform one or more tasks on the microcontroller. Several of the instructions are described below. These instructions can be invoked in special circumstances by, for example, a switch or other equivalent measure.

[0251] The "PUSHL" instruction pushes an 8-bit literal value onto the stack. The exact syntax of the PUSHL instruction is "PUSHL k" where 0 <= k <= 225. Invocation of the command does not affect the status of the microcontroller. The encoding for the PUSHL instruction is "1110 1010 kkkk kkkk" where the 8-bit literal is designated by the kkkk kkkk portion of the instruction. Here, the 8-bit literal is copied to the location that the second file select register ("FSR2") is addressing and then the FSR2 is decremented.

[0252] The "SUBFSR" instruction subtracts a 5-bit literal from a file select register ("FSR"). The syntax for the command is "SUBFSR f, k" where 0 <= f <= 2 and 0 <= k <= 63. Invocation of the command does not affect the status of the microcontroller. The encoding for the command is "1110 1001 ffkk kkkk" which, when invoked subtracts the 6 bit (unsigned) literal from FSRf and store the result back into FSRf where the "ff" portion of the instruction designates the particular file select register, and the "kk kkkk" portion of the instruction designates the literal.

[0253] The "SUBULNK" instruction subtracts a 5-bit literal from the second file select register ("FSR2") and returns the resulting value. The syntax for the command is "SUBULNK k" where 0 <= k <= 63. The invocation of the command does not affect the status of the microcontroller. The encoding for the command is "1110 1001 11kk kkkk" which, when invoked, subtracts the 6 big (unsigned) literal from the FSR2, stores the result back into the FSR2, and returns the result and the execution back to caller.

[0254] The "ADDFSR" instruction adds a 5-bit literal to an FSR. The syntax for the instruction is "ADDFSR f, k" where 0 <= f <= 2 and 0 <= k <= 63. The status of the microntroller is not affected by this instruction. The encoding for the instruction is "1110 1000 ffkk kkkk" and, upon invocation, a 6 bit (unsigned) literal is added to FSRf and the resulting value is stored back into FSRf where the "ff" portion of the instruction designates the particular file select register, and the "kk kkkk" portion of the instruction designates the literal.

[0255] The "ADDULNK" instruction adds a 5-bit literal to the FSR2 and returns the result and the execution back to the calling routine. The syntax for the instruction is "ADDULNK k" where 0 <= k <= 63. The invocation of this instruction does not affect the status of the microcontroller. The encoding for the instruction is "1110 1000 11kk kkkk" where, upon invocation, a 6 bit (unsigned) literal (which is designated by the "kk kkkk" portion of the instruction) is added to the FSR2 and the result is stored back into the FSR2 and execution returned back to caller.

[0256] The "MOVSF" instruction stores a stack location to the general purpose register ("GPR"). The syntax for the instruction is "MOVSF s, d" where 0 <= s <= 127 and 0 <= d <= 4095. The invocation of the instruction does not affect the status of the microcontroller. The encoding for the instruction is in two words, where Word 1 is "1110 1011 0sss ssss" and Word 2

is "1111 dddd dddd dddd" and where the "sss ssss" portion of the first word of the instruction designates a source, and the "dddd dddd dddd" portion of the second word designates the destination. Upon invocation of the instruction, the 7-bit literal value s is added to the value in the FSR2, resulting in the source address of an 8-bit value that is subsequently copied to a location that is designated by the 12-bit value d. The FSR2 value is unaffected by the MOVSF instruction, nor is the status of the microcontroller affected.

[0257] The "MOVSS" instruction copies a stack location to another stack location. The syntax for the instruction is "MOVSS s, d" where 0 <= s <= 127 and 0 <= d <= 127. The invocation of the instruction does not affect the status of the microcontroller. The encoding for the instruction is in two words, where Word 1 is "1110 1011 1sss ssss" and Word 2 is "1111 xxxx xddd dddd" and where the "sss ssss" portion of the first word of the instruction designates a source, and the "ddd dddd" portion of the second word designates the destination. Upon invocation of the instruction, the 7-bit literal value s is added to the value in the FSR2, resulting in the source address of an 8-bit value that is subsequently copied to a location that is designated by an address, that address being determined by adding the 7-bit value d to the value in the FSR2. The FSR2 values is unaffected by the MOVSS instruction, nor is the status of the microcontroller affected.

[0258] The "CALLW" instruction is an indirect call. The syntax for the instruction is "CALLW". Invocation of the instruction does not affect the status of the microcontroller. The encoding for the command is "0000 0000 0001 0100" and, upon invocation of the instruction, the address of the next instruction is pushed onto the hardware stack. Specifically, the values from a first register, such as PCLATU:PCLATH are copied into the upper 16 bits of the program

counter ("PC") and the value in a second register, such as the w register ("WREG"), are copied into the lower 8 bits of the PC.

[0259] The present invention, therefore, is well adapted to carry out the objects and attain the ends and advantages mentioned, as well as others inherent therein. While the present invention has been depicted, described, and is defined by reference to particular preferred embodiments of the invention, such references do not imply a limitation on the invention, and no such limitation is to be inferred. The invention is capable of considerable modification, alternation, and equivalents in form and function, as will occur to those ordinarily skilled in the pertinent arts. The depicted and described preferred embodiments of the invention are exemplary only, and are not exhaustive of the scope of the invention. Consequently, the invention is intended to be limited only by the spirit and scope of the appended claims, giving full cognizance to equivalents in all respects.